2.1 Python's Built-In Functions

In the previous chapter, we began our study of programming in Python by studying three main ingredients: literals, operators, and variables. We can express complex computations using just these forms of Python code, but as the tasks we want to perform grow more complex, so too does the code we need to write. In this chapter, we'll learn about using *functions* in Python to organize our code into useful logical blocks that can be worked on separately and reused again and again in our programs.

Review: Functions in mathematics

Before looking at functions in Python, we'll first review some of the mathematical definitions related to functions from the First-Year CS Summer Prep.

Let A and B be sets. A **function** $f : A \to B$ is a mapping from elements in A to elements in B. A is called the **domain** of the function, and B is called the **codomain** of the function.

Functions can have more than one input. For sets $A_1, A_2, \ldots, A_k$ and B, a k-ary function $f : A_1 \times A_2 \times \cdots \times A_k \to B$ is a function that takes k arguments, where for each i between 1 and k, the i-th argument of f must be an element of A_i, and where f returns an element of B. We have common English terms for small values of k: unary, binary, and ternary functions take one, two, and three inputs, respectively. For example, the addition operator $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a *binary* function that takes two real numbers and returns their sum. For readability, we usually write this function as $x + y$ instead of $+(x, y)$.

Python's built-in functions in Python

We've seen that Python has many operators, like `+` that can be used on various data types. These operators are actually functions represented by symbols (e.g., addition through the `+` symbol). But there aren't enough symbols to represent every function we could ever want. So Python also defines several functions that we can use to perform additional operations; these functions are called **built-in functions**, as they are made automatically available to us anywhere in a Python program. For example, the Python function `abs` takes a single numeric input and returns its absolute value. But how do we actually use it?

A Python expression that uses a function to operate on a given input is called a **function call**, and has the same syntax as mathematics: `<function_name>(<argument>, <argument>, ...)`. For example, here are two examples of a function call expressions that call `abs`:

```
>>> abs(-10)  # Returns the absolute value of -10.
10
```

```
>>> abs(100)
100
```

Function calls are central to programming, and come with some new terminology that we'll introduce now and use throughout the next year.

- In a function call expression, the input expressions are called **arguments** to the function call. *Example*: in the expression `abs(-10)`, the `-10` is the argument of the function call.
- When we evaluate a function call, we say that the arguments are **passed** to the function. *Example*: in `abs(-10)`, we say that `-10` is passed to `abs`.
- When the function call produces a value, we say that the function call **returns** a value, and refer to this value as the **return value** of the function call expression. *Example*: the return value of `abs(-10)` is `10`.

In your mathematical studies so far, you've mainly studied *unary numeric* functions, i.e., functions that take in just one numeric argument and return another number.[1] In

[1] Examples include the sin and log functions.

programming, however, it is very common to work with functions that operate on a wide variety of data types, and a wide number of arguments. Here are a few examples of built-in functions that go beyond taking a single numeric argument:

- The `len` function takes a string or collection data type (e.g., `set`, `list`) and returns the size of its input.[2]

[2] While we defined "size" of these data types back in Section 1.1, we didn't cover them in Python in the last chapter because we were waiting to get to functions.

```
>>> len({10, 20, 30})
3
>>> len('')
0
>>> len(['a', 'b', 'c', 'd', 'e'])
5
>>> len({'David': 100, 'Mario': 0})
2
```

- The `sum` function takes a collection of numbers (e.g., a `set` or `list` whose elements are all numbers) and returns the sum of the numbers.

```
>>> sum({10, 20, 30})
60
```

```
>>> sum([-4.5, -10, 2, 0])
-12.5
```

- The `sorted` function takes a collection and returns a `list` that contains the same elements as the input collection, sorted in ascending order.

```
>>> sorted([10, 3, 20, -4])
[-4, 3, 10, 20]
>>> sorted({10, 3, 20, -4})  # Works with sets, too!
[-4, 3, 10, 20]
```

- The `max` function is a bit special, because there are two ways it can be used. When it is called with two or more inputs, those inputs must be numeric, and in this case `max` returns the largest one.

```
>>> max(2, 3)
3
>>> max(-2, 10, 3, 0, 1, 7)
10
```

  But `max` can also be called with just a single argument, a non-empty collection of numbers. In this case, `max` returns the largest number in the collection.

```
>>> max({2, 3})
3
>>> max([-2, 10, 3, 0, 1, 7])
10
```

- The `range` function we saw in the last chapter takes in two integers `start` and `stop` and returns a value representing a range of consecutive numbers between `start` and `stop - 1`, inclusive. For example, `range(5, 10)` represents the sequence of numbers `5`, `6`, `7`, `8`, `9`. If `start >= end`, then `range(start, end)` represents an *empty* sequence.

One special function: `type`

The last built-in function we'll cover in this section is `type`, which takes *any* Python value and returns its type. Let's check it out:[3]

[3] The term `class` that you see returned here is the name Python uses to refer to mean "data type". More on this later.

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type('David')
<class 'str'>
>>> type([1, 2, 3])
<class 'list'>
>>> type({'a': 1, 'b': 2})
<class 'dict'>
```

If you're ever unsure about the type of a particular value or variable, you can always call `type` on it to check!

Nested function calls

Just like other Python expressions, you can write function calls within each other, or mix them with other kinds of expressions.

```
>>> max(abs(-100), 15, 3 * 20)
100
```

However, just as we saw with deeply nested arithmetic expressions earlier, too much nesting can make Python expressions difficult to read and understand, and so it is a good practice to break down a complex series of function calls into intermediate steps using variables:

```
>>> v1 = abs(-100)
>>> v2 = 15
>>> v3 = 3 * 20
>>>> max(v1, v2, v3)
100
```

Methods: functions belonging to data types

The built-in functions we've studied so far all have one interesting property in common: they can all be given arguments of at least two different data types: for example, `abs` works with both `int` and `float`, `len` and `sorted` work with `set` `list` (and others), and `type` works with values of absolutely any data type. In fact, this is true for almost all built-in functions in Python, as part of the design of the language itself.

However, Python's data types also support operations that are specific that that particular data type: for example, there are many operations we can perform on strings that a specific to textual data, and that wouldn't make sense for other data types.

Python comes with many functions that perform these operations, but handles them a bit differently than the built-in functions we've seen so far. A function that is defined as part of a data type is called a **method**.[4] All methods are functions, but not all functions are

[4] The terms *function* and *method* are sometimes blurred in programming, particularly from language to language, but for us these terms have precise and distinct meanings!

methods. For example, the built-in functions we looked at above are all *not* methods. We refer to functions that are not methods as **top-level functions**. We'll see later how we define functions and methods in Python, but for now let's look at a few examples of methods.

One `str` method in Python is called `lower`, and has the effect of taking a string like `'David'` and returning a new string with all uppercase letters turned into lowercase: `'david'`. To call this method, we refer to it by first specifying the name of the data type it belongs to (`str`), followed by a period (`.`) and then the name of the method.

```
>>> str.lower('David')
'david'
```

Here are a few other examples of methods for different data types, just to give you a sense of the kinds of operations that are allowed.

```
>>> str.split('David wuz hear')           # str.split splits a string into
        words
['David', 'wuz', 'hear']
>>> set.union({1, 2, 3}, {2, 10, 20})     # set.union performs the set union
        operation
{1, 2, 3, 20, 10}
>>> list.count([1, 2, 3, 1, 2, 4, 2], 2)  # list.count counts the number of
        times a value appears in a list
3                                         # (remember, a list can have
        duplicates!)
```

References

- CSC108 videos: Functions (Part 1, Part 2, Part 3)
- Appendix A.1 Python Built-In Function Reference
- Appendix A.2 Python Built-In Data Types Reference (includes descriptions of methods)